National Tsing Hua University
Fall 2023 11210IPT 553000
Deep Learning in Biomedical Optical Imaging
Homework 3

AUTHOR ONE[1] 張皓旻

Student ID:112022533

1. Task A: Reduce Overfitting

我想使用增加 dropout 來減少 overfitting，model 的設定如下圖(Fig.1)，只是簡單的在第三個捲積層和 Linear 層之後增加了機率為 60%的 dropout，其他捲積層都沒有改變。

```
class ConvModel(nn.Module):
    def __init__(self):
        super().__init__()

        # 1 channel, and using 3x3 kernels for simplicity, 256*256
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding='same')
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2) # 128*128

        self.conv2 = nn.Conv2d(32, 32, kernel_size=3, stride=1, padding='same') # 128*128
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2) # 64*64

        self.conv3 = nn.Conv2d(32, 32, kernel_size=3, stride=1, padding='same') # 64*64

        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2) # 32*32

        # Adjust flattened dimensions based on the output size of your last pooling layer
        flattened_dim = 32 * 32 * 32

        self.fc1 = nn.Linear(flattened_dim, 32)

        self.fc2 = nn.Linear(32, 1)
    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool1(x)

        x = F.relu(self.conv2(x))
        x = self.pool2(x)

        x = F.relu(self.conv3(x))

        x = self.pool3(x)
        # Flatten the output for the fully connected layers
        x = x.reshape(x.size(0), -1)  # x.size(0) is the batch size
        x = F.relu(self.fc1(x))



        return self.fc2(x)
```

```
class ConvModel(nn.Module):
    def __init__(self):
        super().__init__()

        # 1 channel, and using 3x3 kernels for simplicity, 256*256
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding='same')
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2) # 128*128

        self.conv2 = nn.Conv2d(32, 32, kernel_size=3, stride=1, padding='same') # 128*128
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2) # 64*64

        self.conv3 = nn.Conv2d(32, 32, kernel_size=3, stride=1, padding='same') # 64*64
        self.droupout3 = nn.Dropout(0.6)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2) # 32*32

        # Adjust flattened dimensions based on the output size of your last pooling layer
        flattened_dim = 32 * 32 * 32

        self.fc1 = nn.Linear(flattened_dim, 32)
        self.droupout = nn.Dropout(0.6)
        self.fc2 = nn.Linear(32, 1)
    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool1(x)

        x = F.relu(self.conv2(x))
        x = self.pool2(x)

        x = F.relu(self.conv3(x))
        x = self.droupout3(x)
        x = self.pool3(x)
        # Flatten the output for the fully connected layers
        x = x.reshape(x.size(0), -1)  # x.size(0) is the batch size
        x = F.relu(self.fc1(x))

        x = self.droupout(x)

        return self.fc2(x)
```

Fig.1 :左圖為原始 model，右圖是更改後，紅色箭頭指出了調整的地方

除了 model 的調整，其他訓練參數都保持一致 (BCEWithLogitsLoss、batch_size=32、epochs=30、optimizer=Adam(lr=1e-3)、StepLR(step_size=10, gamma=0.1))。

Task A 結果:

如下圖所示，Fig.2 是調整 model 前的性能，train acc (Training Accuracy)到達了 100%的同時 val acc (Validation Accuracy)維持在 95%左右不再變化，loss 也是相同的情況，train loss (Training loss)非常低，但 val loss (Validation loss)維持在 0.3 左右不再波動，overfitting 的情況非常嚴重，Fig.3 是 model 調整後的性能表現，train

acc 與 val acc 最終趨於相同，接近 97%，train loss 與 val loss 的趨勢與數值也都十分接近，大約 0.1，可以看出更改後的 model 很好的改善了 overfitting 的情況，可以說這是更 general 的 model，更試用於各種情況，而調整前的 model 則與 Training data 完美的對上了，但對於沒看過的情況，就沒那麼準確了，可以說，調整前的 model 不夠 general。

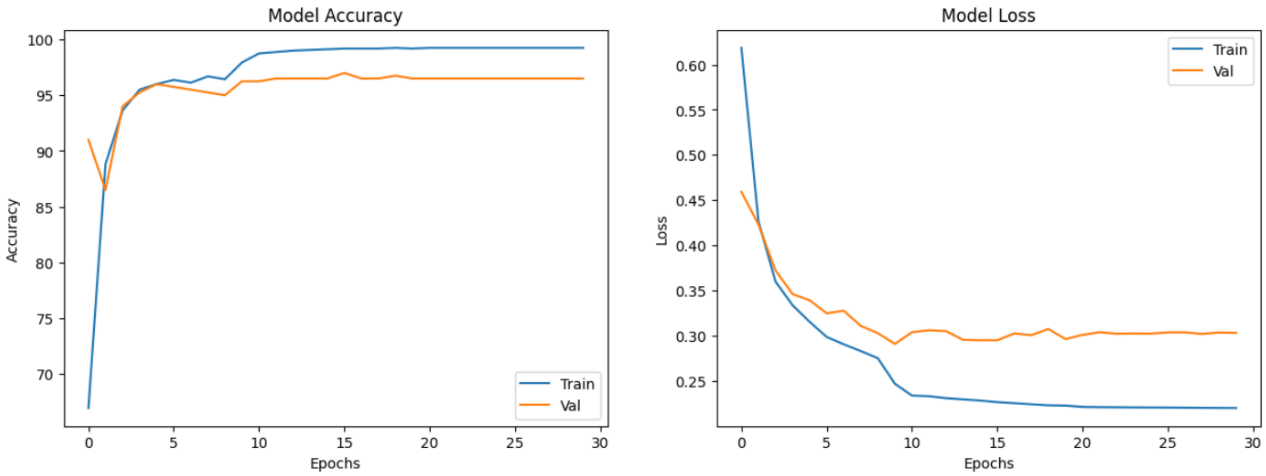

Fig.2 :調整前 model 性能表現

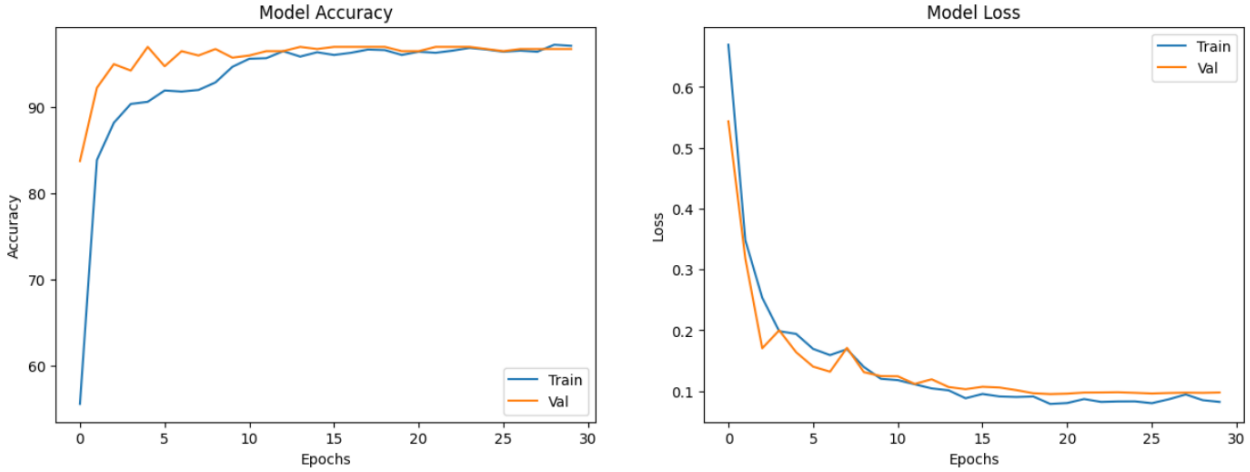

Fig.3 :調整後 model 性能表現

至於 Dropout 為何可以有效防止 overfitting 的發生，我的理解是，由於 model 每次學習時，參數都有機會被歸零，所以 train acc 會不斷地更改，無法找到一種完美符合 Training data 的參數，藉此只好找出一種更 general 的參數，以達到更高的 train acc。

2. Task B: Performance Comparison between CNN and ANN

我想使用上面調整前的 CNN 模型進行比較，目標是建立一個與上述 CNN 規模相同的 ANN 來進行比較，這樣感覺比較公平，但我不確定要怎麼比較他們之間的規模，如果 ANN 用與上述 CNN 完全相同的層數，卷積核的數量直接換成神經元的數量，但 CNN 之後的全連接層數量又十分龐大，在 ANN 中就會變成 32 個神經

元連接 32000 以上的神經元，這將導致模型表現很差，如果我改變 ANN 其他層的神經元數量到與 CNN 捲積層計算次數一樣的數字，此 ANN 模型會大到完全放不進記憶體，所以我最後決定就直接比較 ANN 與 CNN 各自表現比較好的模型，如下圖，Fig.4 是此次實驗所使用的 ANN 模型，Fig.5 是 CNN 模型，其他訓練參數均保持一致(BCEWithLogitsLoss、batch_size=32、epochs=30、optimizer=Adam(lr=1e-3)、StepLR(step_size=10, gamma=0.1))。

```python
class LinearModel(nn.Module):
    def __init__(self):
        super().__init__()

        self.net = nn.Sequential(
                    nn.Flatten(),

                    nn.Linear(256*256*1, 512),
                    nn.BatchNorm1d(512),
                    nn.ReLU(),
                    nn.Dropout(0.9),

                    nn.Linear(512, 256),
                    nn.BatchNorm1d(256),
                    nn.ReLU(),
                    nn.Dropout(0.9),

                    nn.Linear(256, 256),
                    nn.BatchNorm1d(256),
                    nn.ReLU(),
                    nn.Dropout(0.8),

                    nn.Linear(256, 1)
                    )
    def forward(self, x):
        x = self.net(x)

        return x
```

Fig.4 :ANN 模型

```python
class ConvModel(nn.Module):
    def __init__(self):
        super().__init__()

        # 1 channel, and using 3x3 kernels for simplicity, 256*256
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding='same')
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2) # 128*128

        self.conv2 = nn.Conv2d(32, 32, kernel_size=3, stride=1, padding='same') # 128*128
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2) # 64*64

        self.conv3 = nn.Conv2d(32, 32, kernel_size=3, stride=1, padding='same') # 64*64

        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2) # 32*32

        # Adjust flattened dimensions based on the output size of your last pooling layer
        flattened_dim = 32 * 32 * 32

        self.fc1 = nn.Linear(flattened_dim, 32)

        self.fc2 = nn.Linear(32, 1)
    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool1(x)

        x = F.relu(self.conv2(x))
        x = self.pool2(x)

        x = F.relu(self.conv3(x))

        x = self.pool3(x)
        # Flatten the output for the fully connected layers
        x = x.reshape(x.size(0), -1)  # x.size(0) is the batch size
        x = F.relu(self.fc1(x))



        return self.fc2(x)
```

Fig.5 :CNN 模型

他們的性能表現分別如下圖，Fig.6 為 ANN 模型的性能表現，其最好的結果為 train acc 約 92%，val acc 約 94%，Fig.7 為 CNN 模型的性能表現，在訓練到 15 次之前就幾乎收斂完成，並且有 overfitting 的現象，其 train acc 為 100%，val acc 約為 97%。

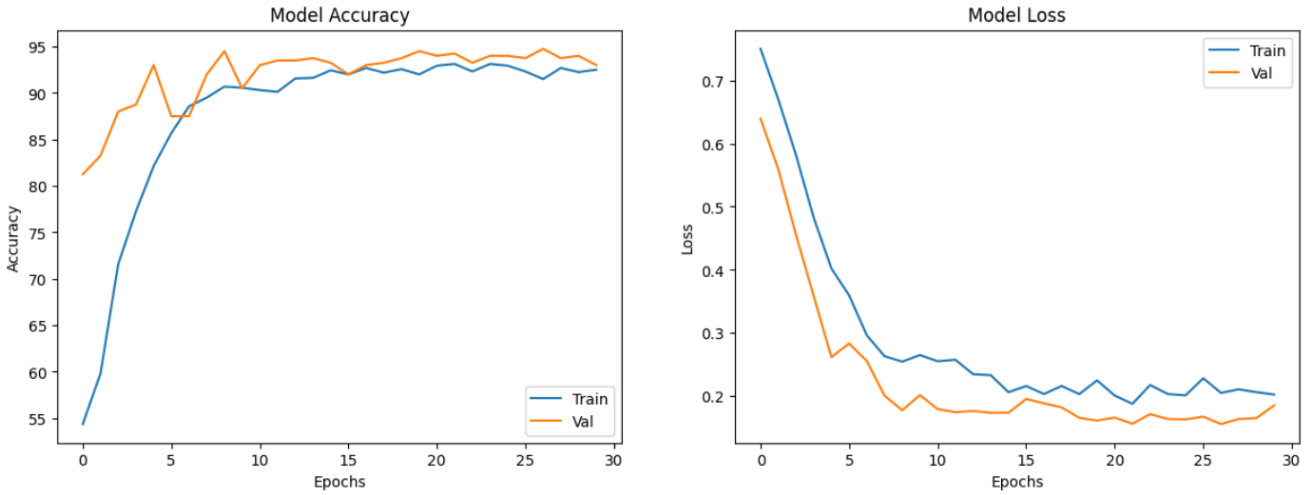

Fig.6 :ANN 模型的性能表現

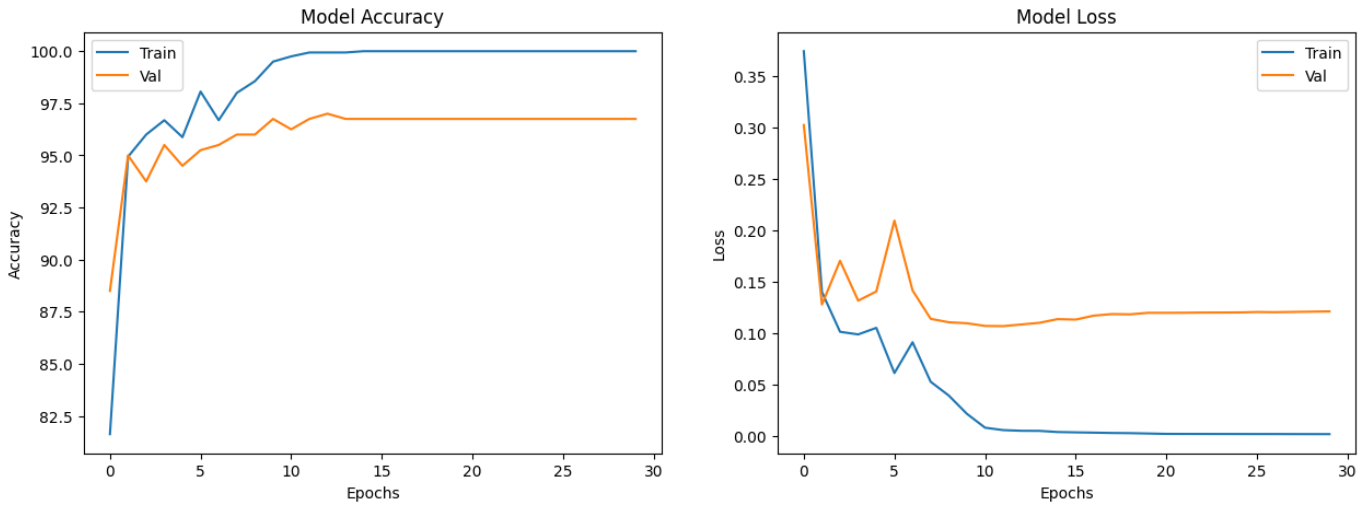

Fig.7 :CNN 模型的性能表現

ANN 模型訓練總時長為 35 秒，CNN 為 1 分 13 秒，感覺 ANN 訓練的比較快，但若今天是使用規模類似的模型，再加上收斂速度的差別，CNN 的速度應該是會快很多，再來正確率的表現也是 CNN 比較好，二維的捲積，更能夠提取出二維空間的特徵，而 ANN 把圖片變成一維數據，相較起來就不太適合用於圖片分析，ANN 更適合處理一維的數據，所以對於生醫影像來說，CNN 模型應該是更好的選擇。

Architecture Description:

如下圖 Fig.8，ANN 架構:先把影像 flatten 成一維，之後接上激活函數和其他 FC 層，最後給出 output; CNN 架構:影像與各種捲積核進行捲積運算，再經過激活函數，然後進行 pooling，提取出各種特徵，最後接上 FC 層，給出 output。

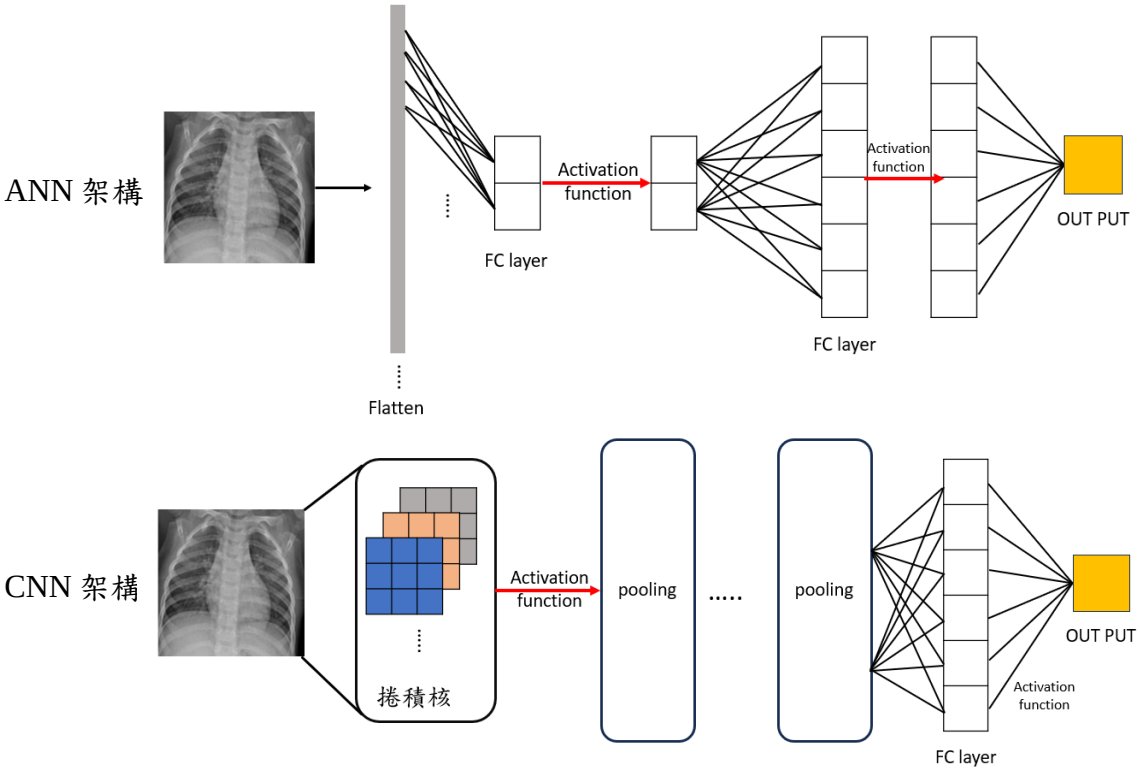

Fig.8 :ANN 與 CNN 架構圖

3. Task C: Global Average Pooling in CNNs

3.1 GAP 的作用:

以 lab4 為例，最後的捲積層 pooling 完之後，會輸出 32 個 32*32 的二維矩陣，GAP 的作用就是把其中每個 32*32 矩陣做平均，並輸出一個值，最後總共輸出 32 個值，剛好把三維的矩陣，降成一維，所以就不用再手動計算之後的神經元要連接幾個輸出，但這也會導致原本有 32*32*32 個神經元連接變得只剩下 32 個神經元連接，這將影響模型的性能。

3.2 提升性能:

我的想法是，由於最後的全連接層所得到的輸入不夠多，導致分類的不夠好，所以我想把最後 pooling 後的輸出增加到 128 個 32*32 的矩陣，所以 GAP 之後會有 128 個輸出，並且之後的全連接層再疊一層，看看能不能提升性能，模型如下圖，Fig.9 是 lab4 原始的模型，Fig.10 是依據上述修改過後的模型，並且保持其他訓練參數都一致 (BCEWithLogitsLoss、batch_size=32、epochs=30、optimizer=Adam(lr=1e-3)、StepLR(step_size=10, gamma=0.1))。

```
class ConvGAP(nn.Module):
    def __init__(self):
        super().__init__()

        self.net = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=3, stride=1, padding='same') ,
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2), # 128*128
            nn.Conv2d(32, 32, kernel_size=3, stride=1, padding='same'), # 128*128
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2), # 64*64
            nn.Conv2d(32, 32, kernel_size=3, stride=1, padding='same'), # 64*64
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2), # 32*32

            nn.AdaptiveAvgPool2d(1),
            nn.Flatten(),
            nn.Linear(32, 1)
        )

    def forward(self, x):

        x = self.net(x)
        return x
```

Fig.9 :lab4 模型

```
class ConvGAP(nn.Module):
    def __init__(self):
        super().__init__()

        self.net = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=3, stride=1, padding='same') ,
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2), # 128*128
            nn.Conv2d(32, 32, kernel_size=3, stride=1, padding='same'), # 128*128
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2), # 64*64
            nn.Conv2d(32, 128, kernel_size=3, stride=1, padding='same'), # 64*64
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2), # 32*32

            nn.AdaptiveAvgPool2d(1),
            nn.Flatten(),

            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, 1)
        )

    def forward(self, x):

        x = self.net(x)
        return x
```

Fig.10 :修改後模型

結果如下圖，Fig.11 是 lab4 模型的性能表現，可以看到 GAP 明顯的降低性能，train acc 與 val acc 都不到 90%，而 Fig.12 是修改後的模型性能表現，可以看到 train acc 與 val acc 都提升了不少，到約 93%，並且沒有 overfitting，不過並沒有恢復到最原本的性能。

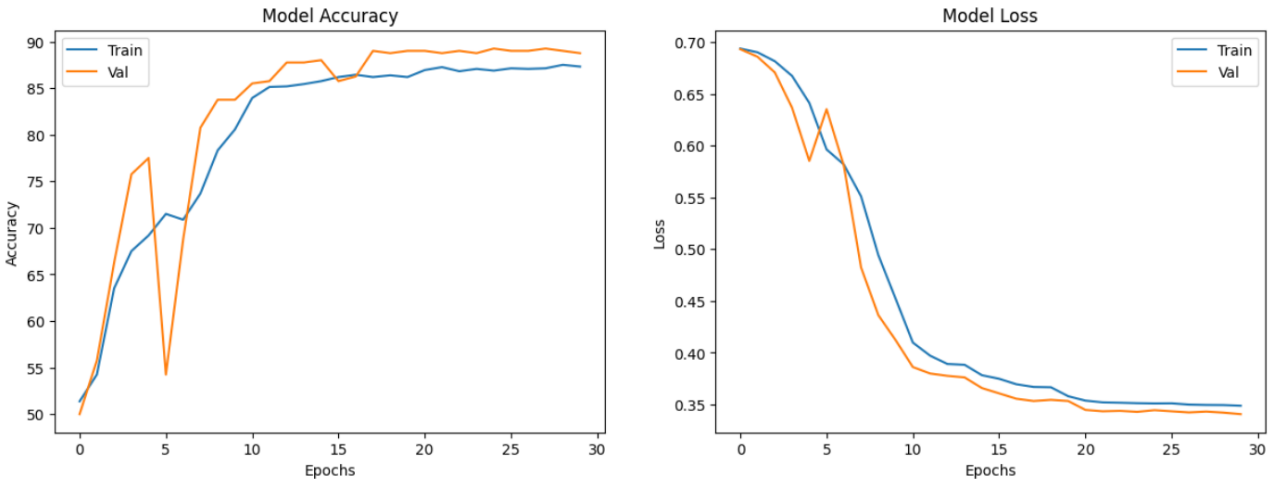

Fig.11 :lab4 模型的性能表現

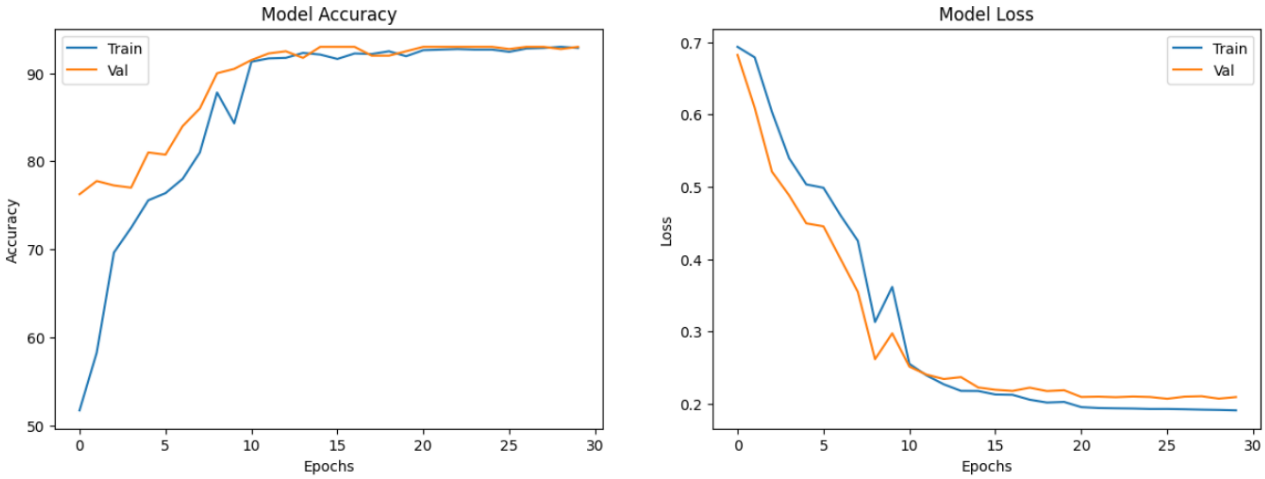

Fig.12 :修改後模型性能表現